THM
TECHNISCHE HOCHSCHULE MITTELHESSEN

Ausarbeitung
Qualitätskontrolle mit dem Smartphone

II2018 Industrie 4.0 - Einführung Konzepte und Technologien

Wintersemester 2022/23

Berkay Oezguer 5338065 Cagkan Arda Sengenc 5337927
Maximilian Biebl 5323481
berkay.oezguer@mni.thm.de
cagkan.arda.sengenc@mni.thm.de
maximilian.biebl@mni.thm.de

Dozent:
Prof. Dr. Überall

22. Februar 2023
Technische Hochschule Mittelhessen, Gießen

Inhaltsverzeichnis

1 Einleitung

Nachfolgend ist unsere Ausarbeitung zu unserem Projekt für das Modul 'II2018 Industrie 4.0 - Einführung Konzepte und Technologien' zu lesen. Das Projekt beschäftigt sich mit der Möglichkeit, eine Qualitätskontrolle mit dem Smartphone durchzuführen. Es wird ein Konzept und die dafür verwendeten Technologien vorgestellt. Im Laufe des Projektes haben wir verschiedene Lösungsansätze für die einzelnen Teilprobleme durchlaufen. Diese Ansätze wie es nicht geht oder nur mit neuen weiteren Problemen sind nachfolgend erläutert. Uns gelang es dabei nicht, eine vollständige App zur Qualitätskontrolle zu implementieren. Wir können aber 2 Prototypen vorzeigen, welche die Möglichkeiten einer Qualitätskontrolle mit dem Smartphone darlegen.

1.1 Ausgangsposition

Qualität ist das Maß dafür, wie gut ein Produkt die Anforderungen erfüllt, die an es gestellt werden[1]. Somit geht es in der Qualitätskontrolle darum, festzustellen, ob ein Produkt seine geforderten Anforderungen erfüllt. Dies kann unterschiedliche Aspekte betreffen, wie zum Beispiel das Aussehen und die Maße des fertigen Produktes. Zum Prüfen, ob die gewünschten Ergebnisse erzielt wurden, müssen daher die entsprechenden Kontrollen durchgeführt werden. In kleineren Unternehmen, die nur geringe Stückzahlen herstellen, müssen diese Kontrollen in der Regel manuell durchgeführt werden. Die Kontrolle, ob ein Produkt das gewünschte Aussehen hat, erfolgt durch eine Betrachtung und möglicherweise einen Vergleich mit einer technischen Zeichnung. Um zu ermitteln, ob ein Produkt den Toleranzmaßen entspricht, können Messschieber oder bei sehr kleinen Toleranzen Mikrometer verwendet werden. Die gemessenen Maße werden anschließend mit der technischen Zeichnung abgeglichen. Im Maschinenbau werden außerdem häufig sogenannte Lehren eingesetzt, die als Schablone dienen, um zu überprüfen, ob ein Produkt der gewünschten Form und den gewünschten Maßen entspricht. Mit einer Lehre lässt sich lediglich bestimmen, ob etwas dem gewünschten Soll entspricht oder nicht. Eine Bestimmung der Abweichung ist nicht möglich. Beispiele hierfür sind eine 90° Winkellehre (Abb. 1) oder Gewindelehren (Abb. 2).

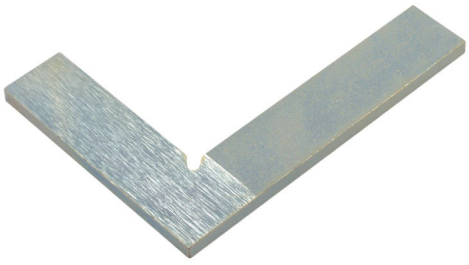

Abbildung 1: 90° Winkellehre
(https://www.dao-ag.de/equipment/media/image/21/aa/c4/article_4048_dao_800x800.jpg)

Abbildung 2: Gewindelehre in der Benutzung
(https://www.dao-ag.de/equipment/media/image/21/aa/c4/article_4048_dao_800x800.jpg)

Der Ablauf einer grundlegenden Qualitätskontrolle ist wahrscheinlich in ähnlicher Form in den meisten Fertigungsunternehmen kleiner bis mittlerer Größe vorhanden. Eines unserer Gruppenmitglieder hat diese so während des Fachabiturs als Industriemechaniker erlebt: Zunächst muss die technische Zeichnung aus einer CAD-Software als Zeichnung exportiert werden. Sie enthält aus jeder Perspektive eine Ansicht des Bauteils, die zur Bestimmung aller Maße notwendig ist (Abb. 3).

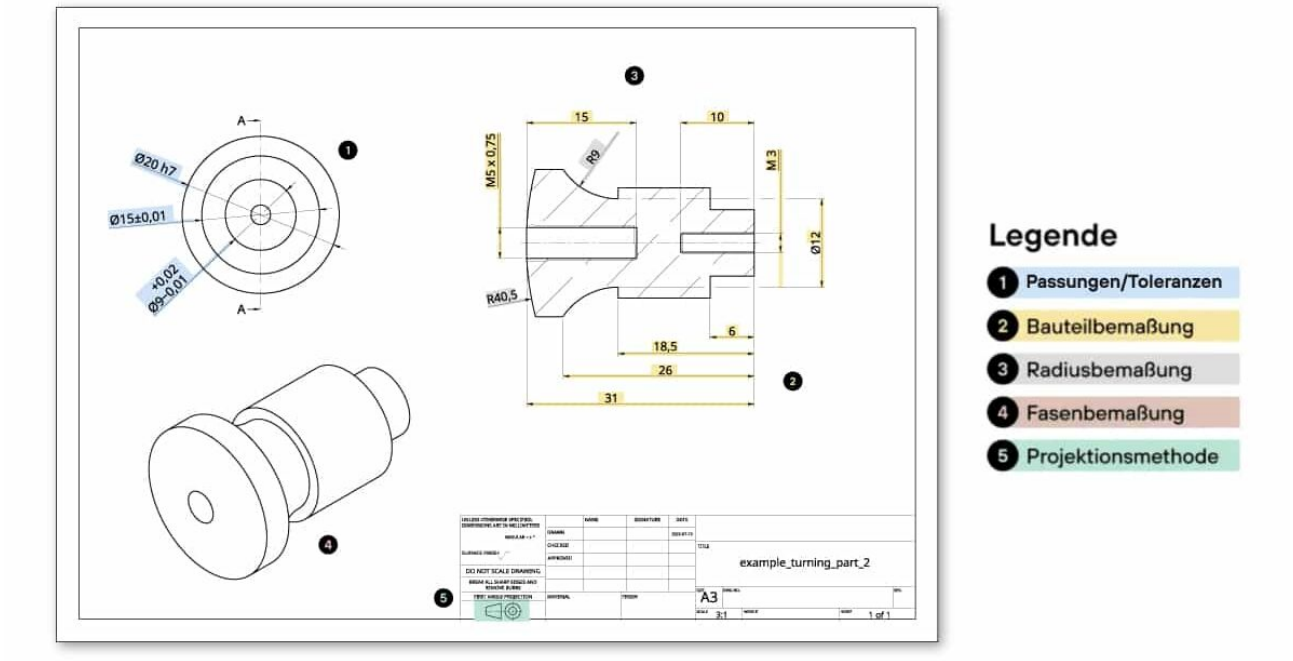

Abbildung 3: Beispiel einer technischen Zeichnung
(https://laserhub.com/app/uploads/2022/08/drehteile-technische-zeichnung-beispiel-e1659356901332.jpg)

Die exportierte technische Zeichnung wird dann in der Regel ausgedruckt und dem Mitarbeitenden übergeben, der die Qualitätskontrolle durchführen soll, dieser macht sich dann mit Messwerkzeug und ggf. einer Lehre an die Kontrolle der Teile. Teile, welche die Qualitätskontrolle nicht bestehen, werden aussortiert oder zur Überarbeitung gegeben. Die grundlegende, konventionelle Qualitätskontrolle hat Probleme und Fehlerquellen. Probleme bei diesem Ablauf:

- Die technische Zeichnung muss gesondert aus der 3D-Zeichnung erstellt werden.
- Bei der Änderungen der 3D-Zeichnung ist die ausgedruckte technische Zeichnung inkonsistent.
- Bei komplexeren Bauteilen mit vielen unterschiedlichen Maßen muss der Mitarbeitende wiederholt auf die Zeichnung gucken.
- Die Dokumentation des Ausschusses muss nochmal in gesonderter Form stattfinden.
- Wenn eine Lehre verwendet wird, muss diese gekauft oder angefertigt werden.

1.2 Zielsetzung

Mit unserem Projekt und unserer Ausarbeitung möchten wir einen Prototyp vorzeigen, der die Möglichkeiten darlegt, wie die grundlegende, konventionelle Qualitätskontrolle in kleinen Fertigungsunternehmen im Sinne der digitalen Transformation in Industrie 4.0 überführt werden kann. Die Qualitätskontrolle soll mithilfe des Smartphones durchgeführt werden. Basierend auf einer 3D-Datei und einem gefertigten Produkt soll überprüft werden, ob das Produkt mit der 3D-Datei übereinstimmt. Bei unserem Projekt beschränken wir uns exemplarisch auf Produkte, die mit einem 3D-Drucker hergestellt werden. Zur Erkennung des Produktes wollen wir die Kamera des Smartphones in Kombination mit der OpenCV Programmbibliothek nutzen.

1.2.1 Rahmenbedingungen

Nach einer grundlegenden Recherche zum Thema haben wir uns Rahmenbedingungen festgelegt:

1. Wir gehen von einem homogenen Hintergrund mit kaum oder keinen Störfaktoren aus.
2. Wir gehen von einem guten Kontrast zwischen Objekt und Hintergrund aus.
3. Um die Maße von einem Objekt zu bestimmen, werden wir einen Aruco oder etwas Vergleichbares als Größenreferenz nehmen.
4. Wir möchten uns bei dem Dateiformat der 3D-Datei auf Eins beschränken.

1.2.2 Meilensteine

Um die Komplexität des Projektes zu reduzieren, haben wir uns zunächst dafür entschieden, etappenweise vorzugehen. Diese groben Meilensteine untergliedern sich:

1. Zunächst möchten wir zu verschiedenen Technologien, die für das Projekt relevant sein könnten, recherchieren und uns mit diesen auseinandersetzen, um ein Grundverständnis für das Projekt zu bekommen.
2. Eine auf dem Smartphone lauffähige App, die Zugriff auf das aktuelle Kamerabild hat.
3. Die OpenCV Bibliothek ist in das Projekt eingebunden und wir können damit beginnen das Kamerabild auszuwerten.
4. Es ist möglich, einfache 2D Formen und deren Größe zuerkennen:
   a) Unterschiedliche Formen wie Kreise, Dreiecke und Quadrate können erkannt werden.
   b) Es kann ein Rechteck und dessen Größen erkannt werden.
5. Es ist möglich, 3D Formen und deren Größe zuerkennen:
   a) Würfel (Abb. 27 im Anhang) erkennen, mit Maßen. Hierbei gehen wir davon aus, dass der Würfel das einfachste Objekt sein wird.
   b) Unterschiedlich große Würfel (Abb. 27 und 28 im Anhang)erkennen und unterscheiden können.
   c) Quader (Abb. 26 im Anhang) erkennen, mit Maßen. Mit dem Quader würden wir die Schwierigkeit langsam steigern.
   d) Pyramide (Abb. 25 im Anhang)erkennen, mit Maßen.
   e) 'Brücke' (Abb. 23 im Anhang) erkennen, mit Maßen. Mit dem Brückenbaustein hätten wir erste Berührungen mit Rundungen, somit könnten wir uns dem Problem nähern runde Objekte zuerkennen.
   f) Zylinder (Abb. 29 im Anhang) erkennen, mit Maßen.
   g) 'Ei' (Abb. 24 im Anhang) erkennen, mit Maßen. Ein eiförmiges Objekt hat keine graden Linien und wir gehen davon aus, dass es somit neue Problem hervorbringt, die es zu lösen gilt.
   h) Vielleicht auch schon komplexere Objekte (Abb. 22 im Anhang) mit Maßen erkennen. Das möchten wir uns aber noch offenhalten, abhängig davon, wie weit wir mit dem Projekt vorankommen. Wir möchten zunächst einmal die grundsätzlichen Möglichkeiten erarbeiten.
   i) Die zuvor genannten Objekte erkennen und mit 3D-Datei abgleichen.

2 Konzept der App

Wir haben uns bei unserem App-Konzept von der konventionellen Qualitätskontrolle inspirieren lassen. Deshalb wollen wir die technische Zeichnung und die Idee der Lehre auf die App übertragen. In der Mitte des Kamerabildes soll eine digitale Lehre anhand der 3D-Datei angezeigt werden. Inspiriert von den technischen Zeichnungen, die meist mehrere Perspektiven auf das Bauteil enthalten, soll man die Lehre anpassen können, um das Bauteil aus allen Perspektiven kontrollieren zu können (Abb. 4 Buttons unten links). Der Nutzer soll, nach Wahl der Perspektive, versuchen, die Kamera so auf das Objekt auszurichten, dass es in die Schablone passt. Die Form und die Maße des Objektes sollen dann mit den erwarteten Werten abgeglichen werden. Entspricht alles den Erwartungen, soll dies visuell kenntlich gemacht werden, z.B. durch Umfärben der Lehre in Grün. Wir haben außerdem einen Button vorgesehen, um das Dateiverzeichnis zu öffnen, wo man dann die aktuell verwendete 3D-Datei auswählen kann.

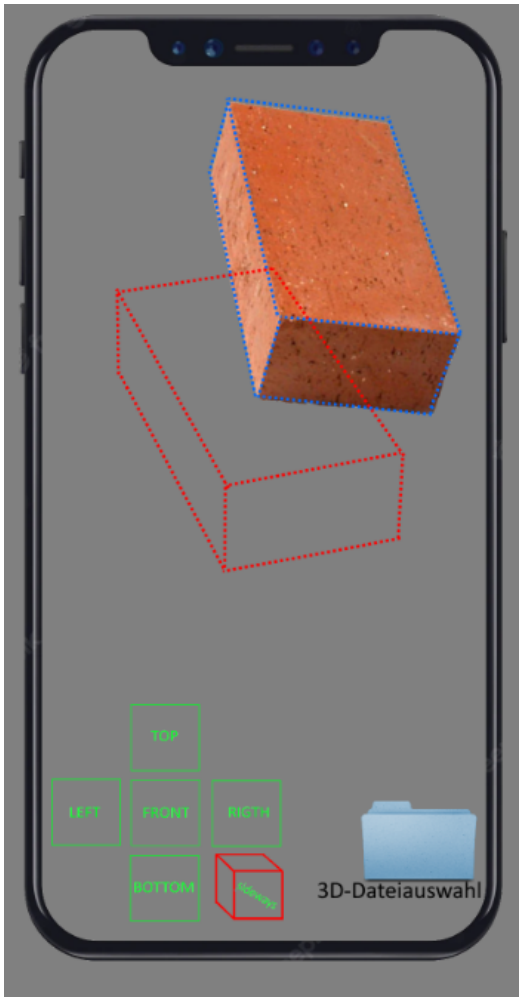

Abbildung 4: App Mockup Bauteil nicht erkannt
(Als vorläufige Skizze mit Paint erstellt.)

- *rot gepunktet: aktuelle Lehre*
- *blau gepunktet: die erkannten Umrisse des Objektes*

Abbildung 5: App Mockup Bauteil erkannt und stimmt mit 3D-Datei überein
(Als vorläufige Skizze mit Paint erstellt.)

- *grün gepunktet: Objekt stimmt mit den Erwartungen überein*

Ergänzbare Features, die aus Zeitgründen nicht Teil des Projektes sind:

- Eine gezielte Führung des Nutzers über die einzelnen Perspektiven, sodass sichergestellt werden kann, das Produkt wurde von allen Seiten kontrolliert.
- Der Abgriff der 3D-Dateien von einem NAS-Server, auf dem der technische Zeichner seine Dateien speichert.
- Ein Backend mit Anbindung an die App, in dem die guten Teile und der Ausschuss dokumentiert werden.

3 Die Wahl der Android Plattform

Zur Entwicklung der App standen ursprünglich die Frameworks Xamarin oder .NET MAUI zur Auswahl. Es gab verschiedene Faktoren, die uns dazu gebracht haben, nach Alternativen zu suchen. Haupteinflussfaktor war die Einfachheit OpenCV mit Kamerazugriff in dem Projekt einzubinden. Nach einer Recherche haben sich für uns zwei Optionen rauskristallisiert.

- Kivy ein Cross-Platform Framework zur Appentwicklung in Python.
- Die native Entwicklung für die Android Platform.

Die Entscheidung, welche Plattform zum Einsatz kommen sollte, war nicht leicht zu treffen. Es dauerte etwa 1-2 Wochen, um die Vor- und Nachteile der einzelnen Plattformen zu bewerten.

3.1 Xamarin/.NET MAUI

Xamarin ist ein Cross-Platform Framework zur Appentwicklung für iOS, Android und Windows. Xamarin basiert auf .NET und die Programmierung findet in C# statt [2]. Bei dem Framework .NET MAUI handelt es sich um eine Weiterentwicklung aus Xamarin.Forms [3]. Zu Beginn des Projektes haben wir viel Zeit damit verbracht, zu beiden Frameworks zu recherchieren. Es gab mehrere Gründe, warum wir uns für keins der beiden Frameworks entschieden haben. Erster Grund dagegen war die Tatsache, dass unsere Gruppe noch keine Erfahrungen mit .NET oder C# hat. Der zweite, wesentlich ausschlaggebendere Grund war die Einbindung von OpenCV mit Kamerazugriff. Bei beiden Frameworks gibt es kaum brauchbare Quellen und wenn wirkte es sehr aufwändig und umständlich.

3.2 Kivy

Kivy ist ein Open Source Framework zur plattformübergreifenden Appentwicklung mit Python.[4] Vorteile von Kivy wären:

- Es wäre Cross-Platform.
- Für OpenCV gibt es viele Tutorials und Beispiele in Python, somit könnten wir uns das Übertragen in eine andere Programmiersprache sparen.
- Während des Projekts hat sich für uns gezeigt, dass Python besser zum Experimentieren und Testen ist, als das streng typisierte Java.

Problem auch hier scheitert es am Kamerazugriff mit OpenCV. Trotz mehrfachem Herumprobieren gelang es uns nicht, das Kameralivebild so abzugreifen, dass wir es mit OpenCV weiter verwerten können. Ein weiterer Nachteil ist die Build-Dauer für Androidgeräte, diese war teils mehrere Minuten, was für ein Projekt welches viel auf Probieren und Analysieren beruht sehr von Nachteil ist.

3.3 Android

Letztendlich haben wir uns für die Entwicklung für Android mit Android Studio entschieden. Ausschlaggebend dafür war:

- Wir können in Java entwickeln, womit jeder in der Gruppe schon Erfahrung hat.
- Es gab geringe Vorerfahrung mit Androidentwicklung.
- Die Einbindung der Kernbibliothek für Android von OpenCV mit Kamerazugriff dauerte es dank YouTube-Video keine 30min bis zur ersten lauffähigen Androidapp[5].

Intern gab es kleinere organisatorische Schwierigkeiten, weil nicht jeder ein zur Entwicklung benötigtes Androidgrät hatte. Das Problem haben wir durch Leihen von zwei Geräten gelöst. In der OpenCV Kernbibliothek für Android ist die benötigte Aruco-Erkennung nicht standardmäßig enthalten, dazu im Abschnitt 4 Bildverarbeitung mehr.

4 Kamerabildanalyse

Zur Erfassung des Ist-Zustands eines zu prüfenden Objektes nutzen wir die Hauptkamera des jeweiligen Smartphones. Die Auswertung des mit der Kamera erfassten Bildes erfolgt mit der open source Bibliothek OpenCV. OpenCV umfasst Funktionen zur Verarbeitung und Analyse von Bildern.

4.1 Maßbestimmung eines Objektes anhand des Kamerabilds

Wie man die Maße eines Objektes anhand des Kamerabildes bestimmt, haben wir uns mit der Hilfe eines YouTube Videos [6] und des zugehörigen Blogeintrags [7] angeeignet. Zunächst haben wir das Tutorial in Python nachprogrammiert, um ein Verständnis für den Code zu bekommen. Wir haben mit dem ganzen noch etwas experimentiert und konnten somit die Funktion der Flächenbestimmung ergänzen. Das größte Problem bei dieser Aufgabe ist die Perspektive. Man kann sich vorstellen, dass sich die Größe eines Objekts ändert, je nach Entfernung zur Kamera. Um dieses Problem zu lösen, haben wir einen Aruco-Marker (siehe Abb. 6) verwendet. Aruco-Marker sind binäre quadratische Marken, ähnlich einem QR-Code, die ursprünglich für Augmented Reality gedacht sind. Die Aruco-Marker nutzen wir für die Schätzung der Kameraposition. Ihr Hauptvorteil ist, dass ihre Erkennung robust, schnell und einfach ist. Das Grundkonzept besteht darin, mit der Hilfe von OpenCV-Funktionen die Konturen des Aruco zu erkennen. Mit dem Erkennen der Arucokonturen gibt uns OpenCV die Möglichkeit auf den Pixelumfang und die Pixelfläche des Arucos zuzugreifen. Bei einem uns bekannten physischen Umfang von 20cm und einer Fläche von $25cm^2$ des Aruco können wir daraus ein Pixel-zu-Zentimeter-Verhältnis errechnen. Dieses Pixel-zu-Zentimeter-Verhältnis ermöglicht es uns, alle Pixelstrecken in Zentimeter umzurechnen.

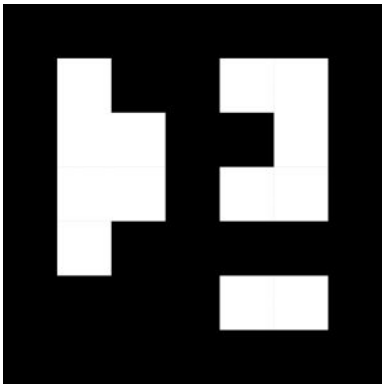

Abbildung 6: Aruco
(https://pysource.com/wp-content/uploads/2021/05/measure-size-of-an-object-with-opencv-aruco-marker-and-python-aruco-tag-300x300.jpg)

4.2 Maßbestimmung Testimplementierung in Python

Da wir Schwierigkeiten hatten, OpenCV inklusive Aruco-Erkennung unter Android zum Laufen zu bekommen, und weil wir zunächst das oben erwähnte Tutorial nachvollziehen wollten, implementierten wir die Maßbestimmung zunächst in Python für den Desktop. Die Implementierung unterteilt sich in zwei Dateien: Eine Hauptdatei und eine weitere Datei namens *object_detector.py* mit der $HomogeneousBgDetector$ Klasse. Die Datei *main.py*: Als Erstes müssen wir den richtigen Aruco-Marker angeben, der unseren Spezifikationen entspricht. Mit der Hilfe einer Variablen namens *aruco_dict* wählen wir die richtige Aruco-Marker-Größe aus. In diesem Fall ist es 5 x 5. Anschließend müssen wir die relevanten Parameter für den Aruco-Marker erstellen und laden. Dazu haben wir eine weitere Variable mit dem Namen *arucoParam* erstellt und diese mit $cv2.aruco.DetectorParameters\_create()$ zugewiesen. Damit haben wir erfolgreich unsere Anpassungsgröße des gewählten Aruco-Markers und die entsprechenden Parameter geladen. Wir legen ein Objekt von $HomogeneousBgDetector$ an und weisen es einer Variable namens *detector* zu. In einer weiteren Variable namens *img* wird das Bild des aktuellen Kameraframes gespeichert. Da wir fortlaufend bis zum Beenden des Programms das Kamerabild auswerten wollen, packen wir den restlichen Code in eine Endlosschleife. Als Nächstes müssen wir die $read()$-Methode aus der Python Bibliothek verwenden, die unser Kamerabild liest und in Tupel umwandelt. Von der Funktion brauchen wir aber die zweite Rückgabe $frame$. $aruco.detectMarkers$ dient der Erkennung von Aruco-Markierungen im Eingabebild. Es werden nur Markierungen gesucht, die in der spezifischen Bibliothek, die wir vorher bekannt gegeben haben, enthalten sind. Die Funktion akzeptiert drei Argumente: unser Kamerabild, die Aruco-Bibliothek und die Aruco-Parameter, die für die Erkennung gebraucht wird. Die Funktion gibt drei Werte zurück, jedoch brauchen wir den ersten Wert, *corners*. In *corners* werden die Koordinaten der 4 Ecken von Aruco-Marker gespeichert. Zweiter Wert wäre die Aruco-IDs der erkannten Marker und als dritter die potenziellen Marker, die gefunden sind, aber der innere Code aus irgendeinem Grund nicht analysiert wird. Die Eckkoordinaten werden dann zu 64-Bit Integer umgewandelt und durch *polyLines* mit grünen Linien gezeichnet, sodass wir sehen können, ob der Marker richtig analysiert wurde (und natürlich wo der Marker ist). Als Nächstes werden alle Kanten der Objekte im Kamerabild gefunden und in die Variable *contours* gespeichert. Wie wir die Objekte erkennen, wird später in *object_detector.py* erklärt. In der großen for-Schleife gehen wir alle gespeicherten Kanten der Objekte durch und speichern in die Variable *cnt*. Mithilfe *minAreaRect* wird ein Quadrat außerhalb des Objekts gezeichnet, damit

wir zentrale Koordinaten- und die Höhe und Breite des Quadrats finden. Dann wird ein roter Punkt in jeweiligen Koordinaten gezeichnet, damit wir später am Rande dieses Punktes auch was schreiben können. Die Variable ist nur für die Fläche der Objekte da, dazu wird später mehr erklärt. Nachdem wir mit *polylines* an allen erkannten Objekten die blauen Linien gezeichnet haben, gibt es eine if-Abfrage, ob Aruco-Marker erkannt wurde. Wenn ja, werden die echten Maße des Markers berechnet. Mit *arcLength* wird der Umfang des Aruco-Markers berechnet und das wird dann durch die realen Maße geteilt. In unserem Fall ist der Marker auf dem Papier 4 mal 4,7 cm groß. Danach wird durch die Pixelfläche auch die reale Fläche berechnet. Mit den ersten beiden *putText* schreiben wir in der Mitte des Objekts, wie hoch und breit es ist. Die letzte *putText* ist für den Text der Fläche da. Wenn aber die if-Anweisung falsch ist – also keinen erkennbaren Aruco-Marker – schreibt das Programm nur pixelweise die Höhe, Breite und die Fläche des Objekts.

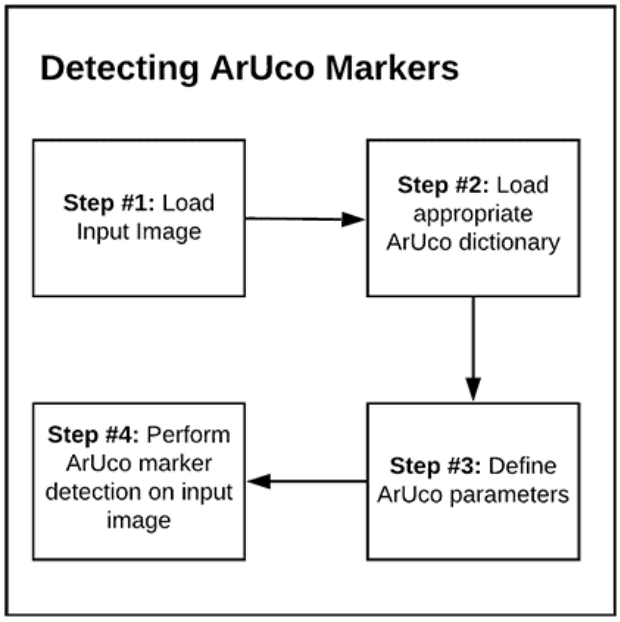

Abbildung 7: Flussdiagramm der Schritte, die zur Erkennung von ArUco-Markern mit OpenCV erforderlich sind.
(https://b2633864.smushcdn.com/2633864/wp-content/uploads/2020/12/aruco_detect_tags_steps.png?size=630x405&lossy=1&strip=1&webp=1)

Die Datei *object_detector.py*: In dieser Datei ist die Klasse *HomogeneousBgDetector* enthalten. Der Zweck dieser Klasse ist, einen Kontrast zwischen dem weißen Hintergrund und den Objekten auf diesem Hintergrund zu schaffen und mithilfe des Kontrasts die Außenform der Objekte zu finden. Erstmal wird die Kamerasicht in Schwarz- bzw. Grau- und Weiß-Form umgewandelt. Dann maskieren wir diese Form und dadurch bekommen wir diesen Kontrast. Mit der Funktion *adaptiveThreshold* finden wir die Flächen, die nicht Weiß sind, also in diesem Fall keine Hintergrundfarbe. Mit der nächsten Funktion $findContours$ werden alle kontinuierlichen Punkte der Flächen herausgefunden, nämlich die Außenform, weil sie die letzte Stelle ist, nach der dann die weiße Farbe anfängt. Die Flächen, die wir durch $findContours$ gefunden haben, müssen auch gespeichert werden. Zur Speicherung brauchen wir erstmal ein leeres Array, nämlich *objects_contours*. Das Problem ist aber, dass das Programm alles bis in das kleinste Detail sieht, wie z.B. die Buchstaben auf einem gedruckten Papier. Da wir solche Kleinigkeiten

nicht brauchen, haben wir eine for-Schleife. In der Schleife kontrollieren wir durch die Funktion $contourArea$ und if-Bedingung, wie groß die kleinste Pixelfläche sein sollte. Wir haben hier die gröbste Fläche für 1 Cent gewählt, also 2000. Wenn die Bedingung passt, fügen wir die Kanten des Objects in unser Array $object\_contours$, was auch unser Returnwert ist.

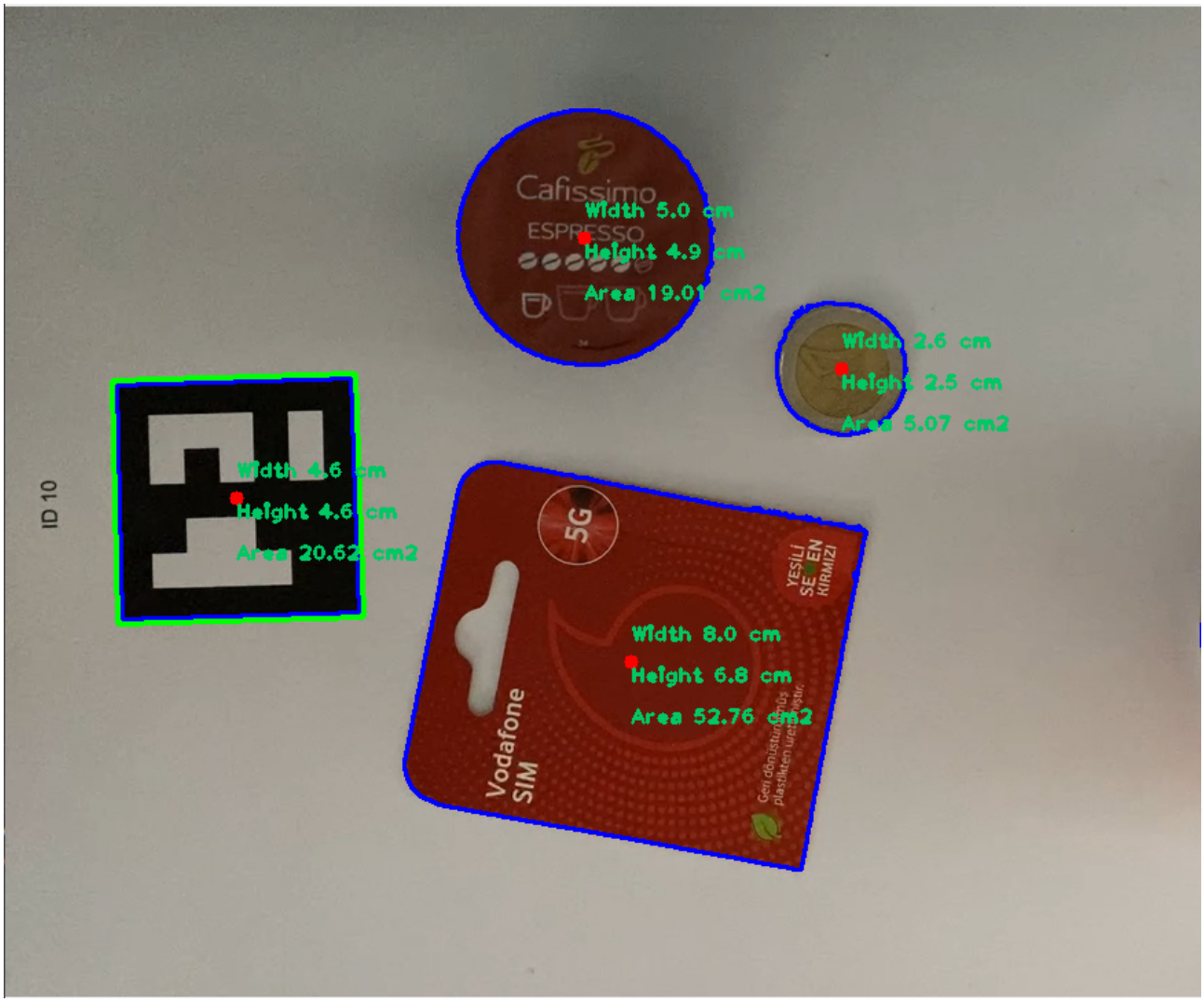

Abbildung 8: Screenshot Maßbestimmung mit Python auf dem Desktop

4.3 Maßbestimmung Implementierung unter Android

Die Nutzung von OpenCV mit Aruco-Erkennung war wie schon erwähnt etwas komplizierter, da diese nicht standardmäßig Teil der OpenCV Implementierung für Android ist. Bei der Aruco-Erkennung handelt es sich um OpenCV-Contrib Modul, welches gesondert entwickelt wird. Nach langen und vielem Suchen, wie man dieses Modul für die Android-Variante von OpenCV einbindet, fanden wir ein GitHub Projekt[8] welches das ganze sehr stark vereinfachte OpenCV-Contrib Module einzubinden. Nachdem wir OpenCV mit Aruco-Erkennung zum Laufen bekommen haben und mit dem Projekt umgezogen sind, konnten wir anfangen, die Erkenntnisse aus Abschnitt 4.2 auf die App zu übertragen. Im Kern funktioniert die Maßerkennung in der App genauso, wir mussten nur die OpenCV Funktionen aus Python in ihr Java äquivalent übersetzen. Beim Übertragen haben wir Logik und Darstellung getrennt.

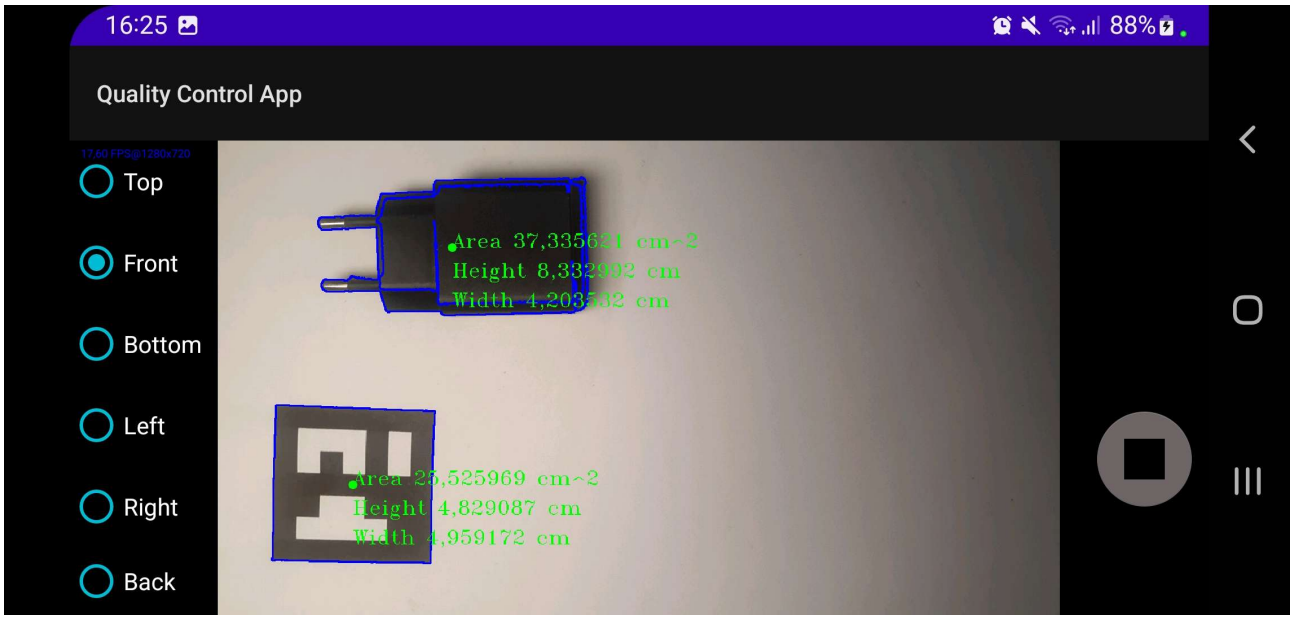

Abbildung 9: Screenshot Maßbestimmung unter Android

5 3D-Dateiauswertung

Bei der 3D-Dateiauswertung wollen wir uns auf ein Dateiformat beschränken. Zunächst haben wir es mit dem *.obj* Dateiformat versucht, als wir damit ins Stocken gerieten, sind wir auf das *.stl* Format umgestiegen. Unsere Grundidee ist, dass wir aus der 3D-Datei ein Bild erstellen und dieses mit OpenCV als Soll-Zustand auswerten. Die Perspektive auf das 3D-Objekt im Soll-Bild soll für die Umsetzung des App-Konzepts (siehe Abschnitt 2) anpassbar sein. Der mit OpenCV ermittelte Soll-Zustand soll dann mit dem Kamerabild als Ist-Zustand verglichen werden.

5.1 Das *.obj* Dateiformat

```
# Exported from 3D Builder
mtllib Wuerfel_40mm.mtl

o Object.1
v -19.999001 -20.000000 39.998001
v 19.999001 -20.000000 39.998001
v -19.999001 20.000000 39.998001
v 19.999001 20.000000 0.000000
v -19.999001 -20.000000 0.000000
v -19.999001 20.000000 0.000000
v 19.999001 -20.000000 0.000000
v 19.999001 20.000000 39.998001

usemtl Green_0
f 1 2 3
f 4 5 6
f 5 4 7
f 8 3 2
f 5 7 2
f 5 3 6
f 8 2 7
f 6 3 8
f 5 1 3
f 7 4 8
f 2 1 5
f 8 4 6
```

Listing 1: .obj am Beispiel des Wuerfel_40mm

In Listing 1 befindet sich ein einfaches Beispiel einer *.obj*-Datei anhand unseres Würfels mit 40mm Kantenlänge. Diese wurde mit dem 3D Builder von Microsoft erstellt. Die Datei beginnt zunächst mit einem optionalen Kommentar, in diesem Fall mit einem vom 3D Builder automatisch generierten. Darauf folgt die optionale Einbindung der Materialbibliothek, darin ist die Textur enthalten. Fortlaufend folgt eine optionale Benennung des Objektes. Anschließend folgen mit v beginnend die Zeilen mit den Ortsvektoren der Eckpunkte. Abschließend folgt das Aufspannen der Textur. Dazu wird erst die Textur angegeben und anschließend die Nummern der zuvor angegebenen Ortsvektoren, zwischen denen die Textur dann aufgespannt werden soll. Im Fall vom 3D Builder werden die Texturen immer zwischen 3 Ortsvektoren aufgespannt.

5.1.1 Erster naiver Versuch

Wir waren zunächst naiv und versuchten das *.obj*-Dateiformat händisch auszuwerten. Der in Abschnitt 5.1 erklärte Aufbau ist zum Glück gut dokumentiert. So haben wir zunächst eine Klasse erstellt, in denen wir die Ortsvektoren speichern können.

```
public class MyCoordinates {
    int x;
    int y;
    int z;

    public MyCoordinates(int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

Listing 2: MyCoordinates Klasse

Nachdem wir die *.obj*-Datei als String eingelesen haben, splitten wir diesen Sting nach den Leerzeichen. Wir können jetzt mit einer einfachen *for*-Schleife und einer *if*-Anweisung die Strings lesen und da wo es den Buchstaben 'v' gibt, können wir die kommenden Zahlen im Konstruktor unserer *MyCoordinates*-Klasse übergeben.

```
        String content = new String(Files.readAllBytes(Paths.get("
    wuerfelpositive.obj")));
        System.out.println(content);
        String[] lines = content.split("\\s+"); //splits String whitespace
    regex
        System.out.println(lines[10]);
        ArrayList<MyCoordinates> myList = new ArrayList<>();
        for (int i = 0; i < lines.length-3; i++) {
            if (lines[i].equals("v")) {
                float temp1 = Float.parseFloat(lines[i+1]);
                float temp2 = Float.parseFloat(lines[i+2]);
                float temp3 = Float.parseFloat(lines[i+3]);
                myList.add(new MyCoordinates((int)temp1, (int)temp2, (int)
    temp3));
            }
        }
```

Listing 3: Einlesen der Vektoren

Wir dachten, damit hätten wir alles, was wir brauchen um das Modell zu visualisieren.
Beim Würfel 10 war es kein Problem zu zeichnen, denn wir brauchen nur 4 Ortsvektoren. Das haben wir probiert, indem wir nur die Elemente aus der Liste ausgewählt haben, die denselben z-Wert haben. Nachdem wir die Elemente gefunden haben, gab es eine Kontrolle mit einer *if*-Anweisung, ob die x-Werte oder y-Werte dieser Elemente gleich sind. In der Abbildung 11 sehen wir, wie ein Quadrat dargestellt wird. Mit der *if*-Anweisung lassen wir die diagonale Linie weg (siehe Abb.12).

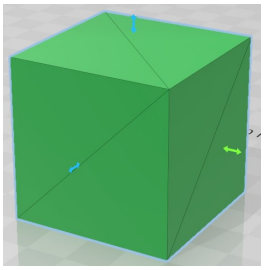

Abbildung 10:

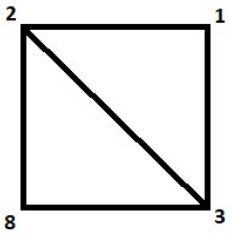

Abbildung 11:

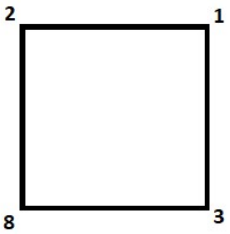

Abbildung 12:

Wenn die kontrollierten Paare den gleichen X-Wert bzw. Y-Wert haben, dann haben wir zwischen diesen beiden Punkten eine Linie zeichnen lassen. Somit haben wir ein Quadrat auf dem Bild. Dieses Verfahren funktioniert nur mit einfachen Objekten, wie z.B. Würfel oder Rechtecken, weil ihre Punkte überschaubar sind. Kugeln oder Zylinder hingegen haben deutlich mehr Punkte als Rechtecke. 3D-Objekte mit Rundungen sind in ihrer Interpretation komplexer, da die Punkte nicht mehr auf einer Ebene liegen und man die Flächen berücksichtigen müsste, um verdeckte Punkte auszuschließen. Wenn man sie so zeichnen würde wie ein Würfel, würde man nicht zum gewünschten Ergebnis gelangen.

5.1.2 Idee mit Processing

Nach den ersten gescheiterten Versuchen händisch die *.obj* auszuwerten, war klar wir brauchen ein Hilfsmittel. Bei unserer Recherche stießen wir dabei auf Processing [9], welches uns schon aus vorherigen Semestern bekannt war. Processing ist eine Programmierbibliothek, die sich an Programmieranfänger richtet. Mit Processing ist es möglich Grafiken, Animationen, Musik und andere Medieninhalte zu erstellen. Ein Processingprogramm besteht im Normalfall aus einer *setup*() Methode und einer *draw*() Methode. Die *setup*() Methode wird einmalig vor der *draw*() Methode aufgerufen, in ihr werden initial Werte wie die Malfläche festgelegt. In der *draw*() Methode wird das, was gemalt werden soll, programmiertechnisch beschrieben. Die *draw*() Methode wird standardmäßig in einer Endlosschleife 60-mal pro Sekunde aufgerufen. Processing stellt mit seiner *loadShape*() Methode die Möglichkeit *.obj* Dateien zu laden und mit *shape*() diese einzuzeichnen. Vorteile von Processing sind, dass es eine hervorragende Dokumentation besitzt, es in Java implementiert ist und sogar eine spezielle Version für Android [10] existiert. Testweise haben wir zunächst in der offiziellen Processing IDE ein einfaches Programm (siehe Listing 4) geschrieben. Das Programm enthält alles, was wir zur Darstellung einer *.obj* für die Verwendung in unserer App bräuchten. Mit dem $PGraphics\ buffer$ nutzen wir die Möglichkeit von Processing etwas außerhalb des Bildschirms zu zeichnen. Die Methoden *translate*() und *rotate*() geben uns die Möglichkeit, das 3D-Objekt zu drehen und zu verschieben. Dadurch könnten wir das 3D-Objekt aus allen Perspektiven mit dem fertigen Produkt vergleichen.

```
private PGraphics buffer;
private PShape obj;
private Perspective currentView = Perspective.FRONT;
private int width = 300;
private int height = 500;

public void settings() {
  size(this.width, this.height, P3D);
}
public void setup() {
  obj = loadShape("Bird.obj");
  buffer = createGraphics(this.width, this.height, P3D);
}
private void drawOffscreen() {
  buffer.beginDraw();
  buffer.background(255);
  buffer.translate(width / 2, height / 2, 300);
  setView(currentView);
  buffer.shape(obj);
  buffer.endDraw();
}
public void draw() {
  drawOffscreen();
  image(buffer, 0, 0);
}
enum Perspective {
  TOP,
    BOTTOM,
    FRONT,
    LEFT,
    RIGHT,
    BACK;
}
public void setView(Perspective p) {
  switch (p) {
  case TOP:
    buffer.rotateX(-HALF_PI);
    break;
  case BOTTOM:
    buffer.rotateX(HALF_PI);
    break;
  case LEFT:
    buffer.rotateY(HALF_PI);
    break;
  case RIGHT:
    buffer.rotateY(-HALF_PI);
    break;
  case BACK:
    buffer.rotateX(PI);
    break;
  case FRONT:
    break;
  }
}
```

Listing 4: Processing Beispiel zum Darstellen einer .obj Datei

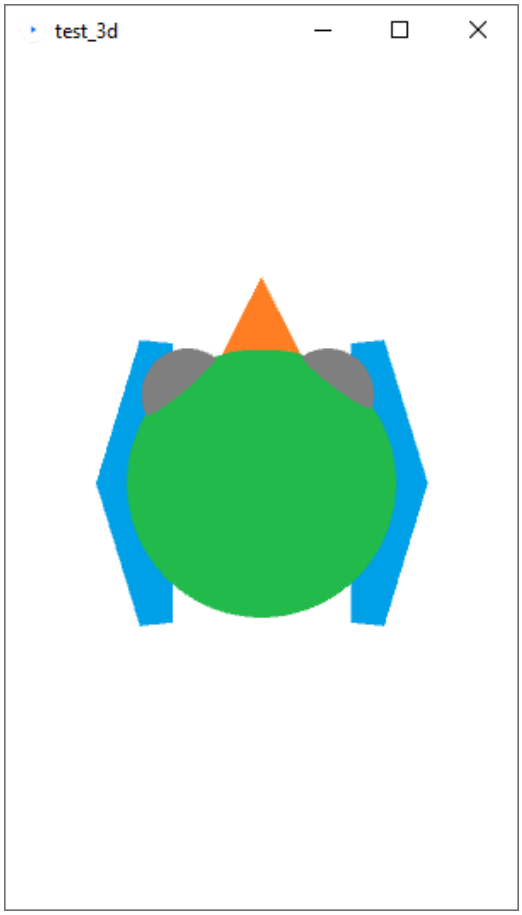

Abbildung 13: currentView = Perspective.FRONT

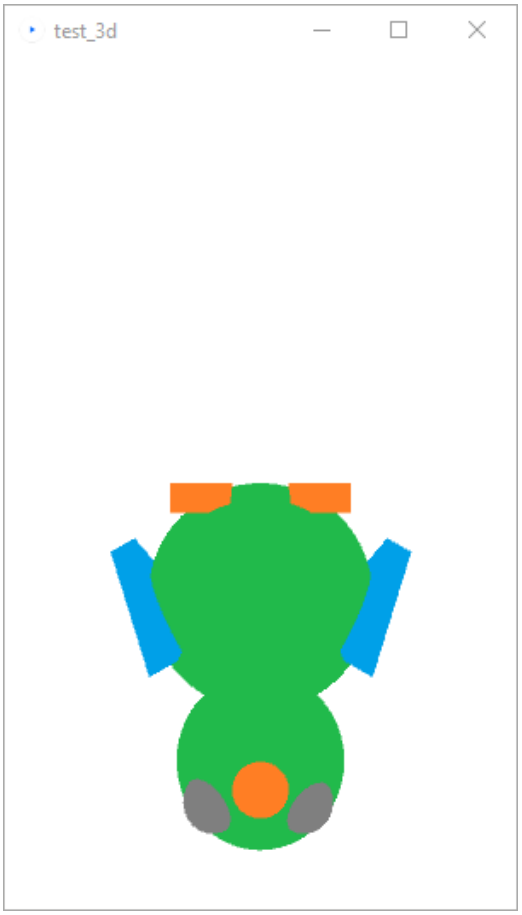

Abbildung 14: currentView = Perspective.TOP

Nach dem Test ging es ans Übertragen auf unser Androidprojekt. Das Einbinden der Processingbibliothek im Projekt ging dank Tutorial [11] schnell und einfach. Wir haben zunächst die *processing − core.jar* von Processing 3 genutzt. Der Contribution Manager von Processing 4 ist aktuell kaputt, dieser ist zum Downloaden der Android Version nötig. Mit einem Workaround, bei dem man ein existierendes Processing-Android-Projekt mit der Processing 4 IDE öffnet, kann man den Download der neusten *processing − core.jar* für Android automatisch hervorrufen. Mit diesem Workaround konnten wir dann die Processingversion im Projekt updaten. In 2D konnten wir auch erfolgreich ein Processingbild außerhalb des Bildschirms ohne die Verwendung der klassischen *draw*() Methode malen. Dieses Bild konnten wir mit der Methode in Listing 5 in eine Matrix für die weitere Verwendung in OpenCV um wandeln. Als OpenCV-Matrix haben wir es dann testweise auf dem Bildschirm anstelle des Kamerabildes wieder gegeben (siehe Abb. 15).
Problem ist das Malen außerhalb des Bildschirms in 3D. Die *createGraphics*() Methode und die *loadShape*() Methode sind in ihrer Verwendung unter Android auf die klassische Verwendung der *draw*() und *setup*() Methoden angewiesen. Unter Android benötigt Processing in seiner klassischen Verwendung eine Android-Activity, wo dann das gemalte Bild dargestellt werden kann. In unserem speziellen Fall existiert diese Activity nicht, da wir das Bild rein im Hintergrund als Soll-Bild benötigen. Wir haben viel Zeit damit verbracht, einen Workaround für dieses Problem zu finden. Leider ohne Erfolg.

```
Mat toMat(PImage image) {
        //source: https://gist.github.com/Spaxe/3543f0005e9f8f3c4dc5
        int w = image.width;
        int h = image.height;
        Mat mat = new Mat(h, w, CvType.CV_8UC4);
        byte[] data8 = new byte[w * h * 4];
        int[] data32 = new int[w * h];
        arrayCopy(image.pixels, data32);
        ByteBuffer bBuf = ByteBuffer.allocate(w * h * 4);
        IntBuffer iBuf = bBuf.asIntBuffer();
        iBuf.put(data32);
        bBuf.get(data8);
        mat.put(0, 0, data8);
        return mat;
}
```

Listing 5: konvertiert ein Processingbild in eine OpenCV-Matrix

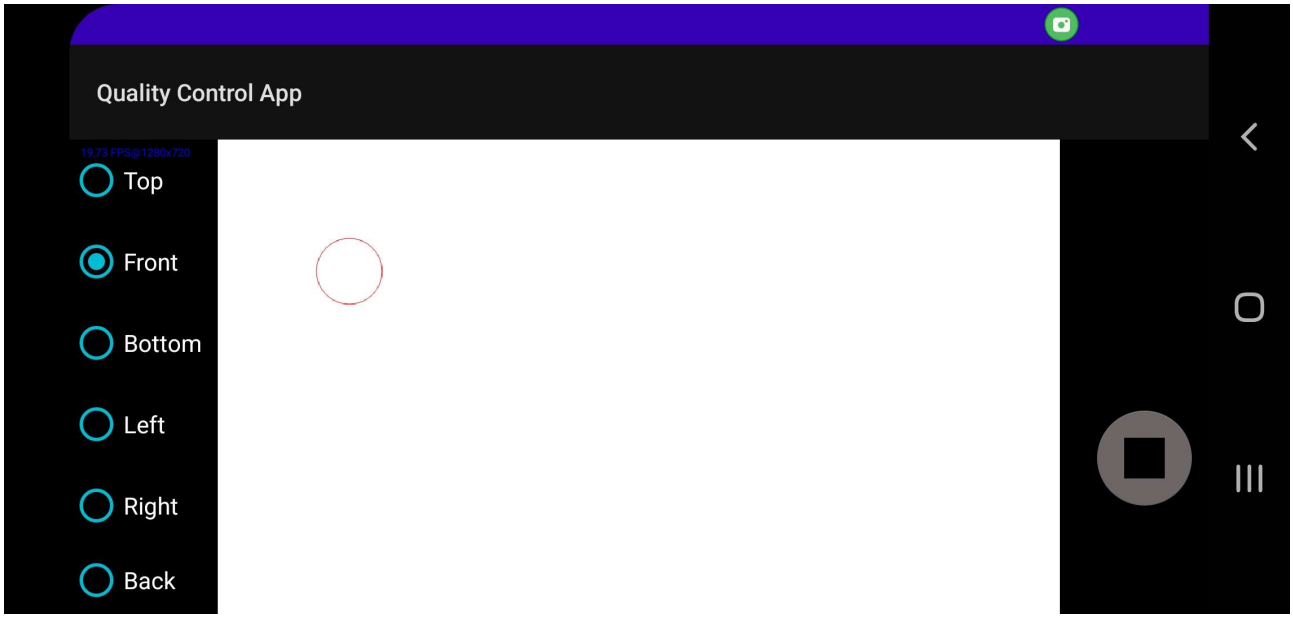

Abbildung 15: Ein mit Processing gemalter Kreis wird als OpenCV Matrix auf dem Bildschirm wiedergegeben.

Eine, unserer Meinung nach, unschöne Lösung ist es eine Activity zu erstellen die zum Zeichnen geöffnet und unmittelbar wieder geschlossen wird. Probleme sind:

- bei jedem Perspektivwechsel diese Activity erneut aufgerufen werden muss
- die schlechte Performance
- häufige Perspektivwechsel führen zum Absturz
- es ist einfach nicht sinnvoll eine Activity zu starten und unmittelbar zu schließen

Mit dieser Lösung ist es zumindest möglich, unser angestrebtes Ziel zu zeigen (siehe Abb. 16 und 17).

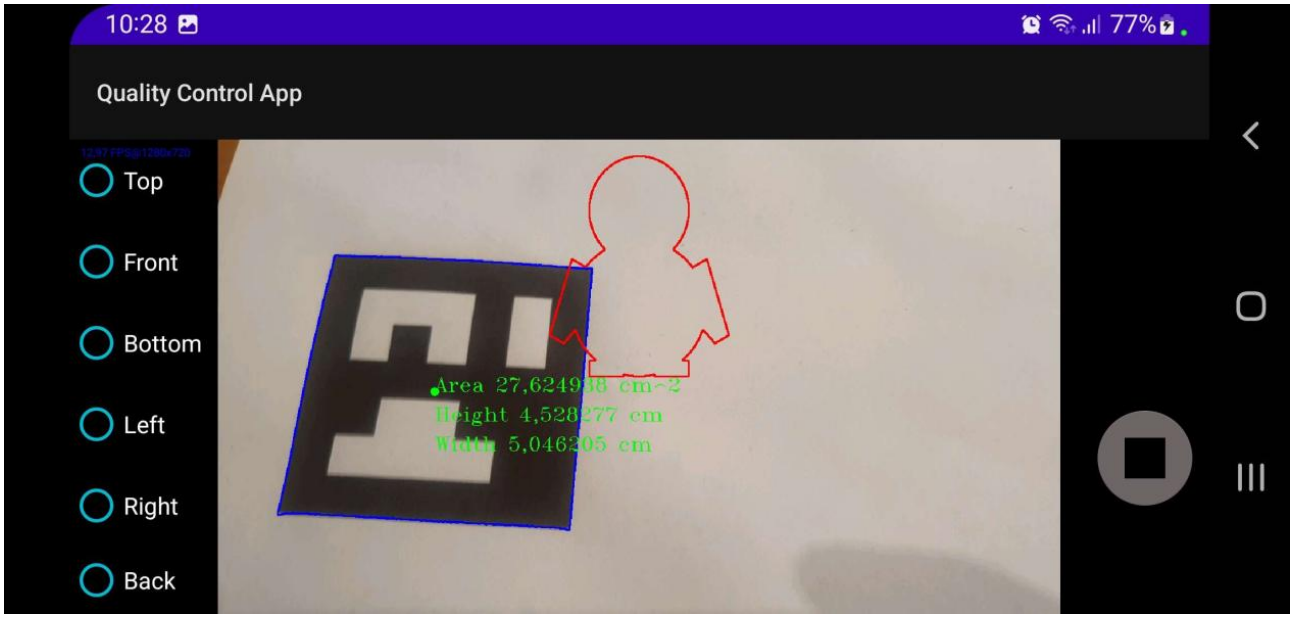

Abbildung 16: Lehre aus .obj-Datei

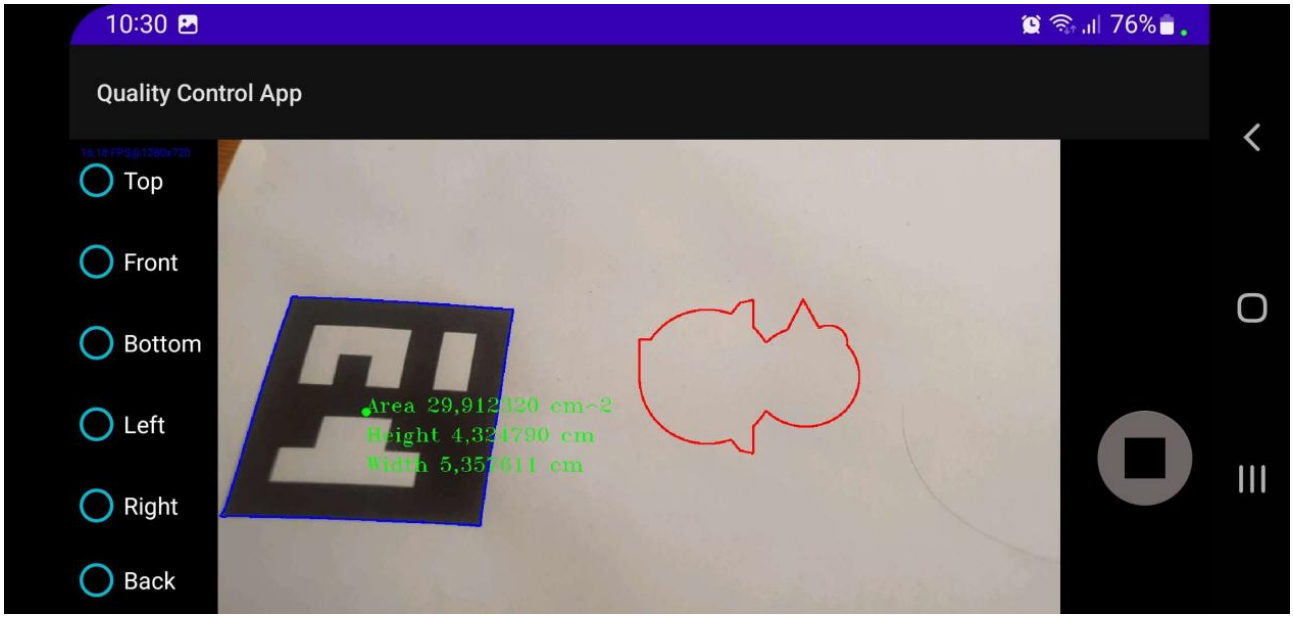

Abbildung 17: Lehre aus .obj-Datei seitlich

5.2 Das *.stl* Dateiformat

Nach den wenig erfolgreichen Versuchen mit dem *.obj*-Dateiformat, kamen wir auf die Idee auf das *.stl*-Dateiformat umzusteigen. Genauso wie in *.obj*-Dateien beinhaltet eine *.stl*-Datei auch die Eckpunkte des Objekts als Ortsvektoren. Verbundene Ortsvektoren werden zwischen den Key-Words *outer loop* und *endloop* angegeben. In Listing 6 sieht man, dass diese Punkte ein Dreieck bilden, genau wie beim *.obj*-Dateiformat.

```
solid Exported from Blender-2.80 (sub 75)
facet normal 0.000000 0.000000 1.000000
outer loop
vertex 1.000000 1.000000 1.000000
vertex -1.000000 1.000000 1.000000
vertex -1.000000 -1.000000 1.000000
endloop
endfacet
facet normal 0.000000 0.000000 1.000000
outer loop
vertex -1.000000 -1.000000 1.000000
vertex 1.000000 -1.000000 1.000000
vertex 1.000000 1.000000 1.000000
endloop
endfacet
facet normal 0.000000 0.000000 -1.000000
outer loop
vertex 1.000000 1.000000 -1.000000
vertex 1.000000 -1.000000 -1.000000
vertex -1.000000 1.000000 -1.000000
endloop
endfacet
facet normal 1.000000 0.000000 0.000000
outer loop
vertex 1.000000 1.000000 -1.000000
vertex 1.000000 1.000000 1.000000
vertex 1.000000 -1.000000 1.000000
endloop
endfacet
facet normal 1.000000 0.000000 0.000000
outer loop
```

Listing 6: .stl Beispiel

Python selbst bietet bereits einige Hilfsmittel, um *.stl*-Dateien zu visualisieren. Wir haben daher eine Auswertung des *.stl*-Format in Python umgesetzt. Die Idee war, die *.stl*-Dateien als mathematische Modelle zeichnen zu lassen und dieses dann als *.png*-Datei zu exportieren.

```
1  class RenderObject:
2      def __init__(self):
3          pass
4
5      def import_object(self, objPath):
6
7          figure = pyplot.figure()
8          axes = figure.add_subplot(projection='3d')
9
10         # Load the STL files and add the vectors to the plot
11         obj_mesh = mesh.Mesh.from_file(objPath)
12
13         axes.add_collection3d(mplot3d.art3d.Poly3DCollection(obj_mesh.
       vectors))
14
15         # auto scale
16         scale = obj_mesh.points.flatten()
17
18         axes.auto_scale_xyz(scale, scale, scale)
19         # axes.view_init(0, 0)  # from the side
20         # axes.view_init(90, 0) # from top or bottom
21         axes.view_init(0, 90) # from front or back
22         pyplot.grid(False)
23         pyplot.axis('off')
24         pyplot.savefig("render.png")
25         # Show the plot to the screen
26         # pyplot.show()
```

Listing 7: RenderObject

Erstmal erstellen wir eine Figur und mit $add\_subplot(projection = \text{`}3d\text{`})$ fügen wir dreidimensionalen Achsen zu der Variable hinzu. In der nächsten Zeile importieren wir die Textur aus der gewünschten *.stl*-Datei zu der Variable *obj_mesh*. Da unsere *.stl*-Datei 3D ist und wir am Anfang auch einer Variable 3D zugewiesen haben, knüpfen wir jetzt Ortsvektoren von der *.stl*-Datei mit unserer 3DVariable 'axes'. Die Funktion $flatten()$ kopiert die dreidimensionalen Ortsvektoren des Modells und macht sie eindimensionale Vektoren, welche der Variable 'scale' zugewiesen werden. Damit die Umgebung für die Visualisierung des Objekts groß genug ist, wird mit $auto\_scale\_xyz(scale, scale, scale)$ die Größe der Umgebung erstellt. Hier in den beiden Bildern 18 und 19 sieht man, dass das Koordinatensystem für das Objekt die richtige Größe hat. Die Funktion *view_init* rotiert das Koordinatensystem, wie wir wollen. In den nächsten drei Zeilen lassen wir den Hintergrund der Visualisierung löschen und das Bild der Visualisierung mit dem Namen *render.png* exportieren (Abb. 20).

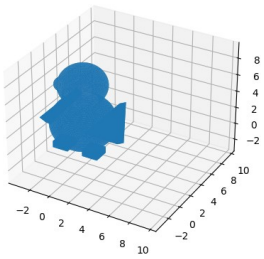

Abbildung 18:

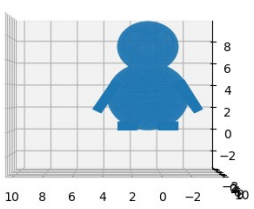

Abbildung 19:

Abbildung 20:

Von diesen gerenderten *.png*-Dateien lassen wir mithilfe von *object_detector.py* die Konturen erkennen und auf dem Kamerabild (Abb.21) wiedergeben. Wie bereits erwähnt wurde, enthält die Visualisierung nicht die Originalgröße des Objekts, daher können wir nur mit Konturen arbeiten.

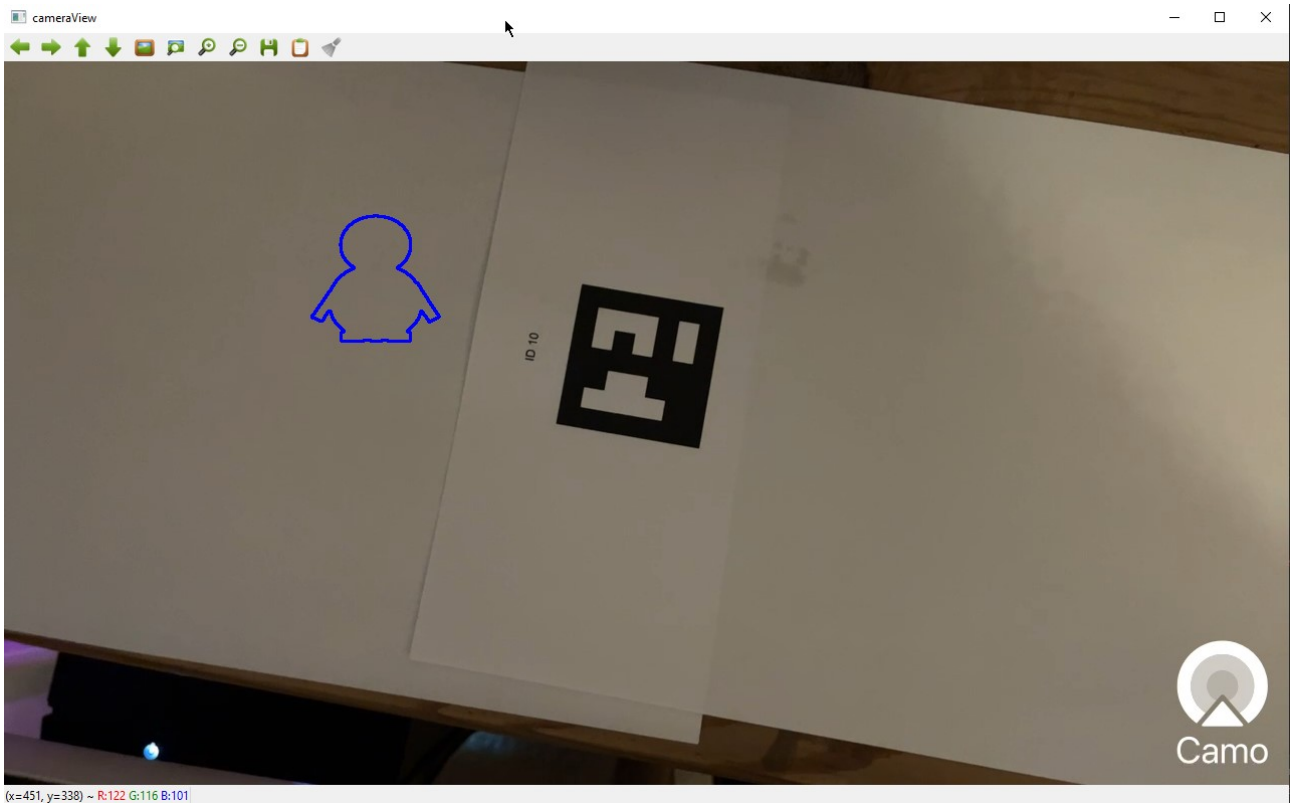

Abbildung 21: Konturen von *bird.stl* auf Kamerabild

6 Fazit

6.1 Zusammenfassung

Wir halten eine Qualitätskontrolle mit dem Smartphone für umsetzbar. Man kann durch die Qualitätskontrolle mit dem Smartphone die in Abschnitt 1.1 aufgezählten Problem umgehen. Die gesonderte Erstellung einer technischen Zeichnung aus der 3D-Datei kann als nicht wertschöpfende Tätigkeit vermieden werden. Bei Änderungen des 3D-Modells muss die Zeichnung nicht erneut ausgedruckt werden, was auf lange Sicht Kosten spart. Die Zeichnung und das 3D-Modell könnten nicht länger inkonsistent werden, wenn in der App die 3D-Datei direkt von einem gemeinsamen Server mit dem technischen Zeichner geholt wird. Der Vorgang der Qualitätskontrolle wäre schneller, da die ausführende Person keine wiederholten abgleiche mit der technischen Zeichnung vornehmen muss. Es muss keine Lehre gekauft oder hergestellt werden, da diese digital ist. Mithilfe von OpenCV und dem Kamerabild ist es möglich, den ist-Zustand eines gefertigten Objektes zu erfassen. Durch einen Aruco ist es möglich ein Pixel-zu-Zentimeter-Verhältnis zu errechnen und so unabhängig von der Perspektive die Maße eines Objektes zu bestimmen. Zur Auswertung einer 3D-Datei sollte ein Framework zur Hilfe gezogen werden, da eine eigene Implementierung zur Auswertung sehr umständlich und zeitaufwändig ist. Wir halten es für sinnvoll aus der 3D-Datei ein Vergleichsbild zu erstellen, was mittels OpenCV mit dem Kamerabild verglichen werden kann.

6.2 Reflexion & Bewertung der Aufgabenstellung

Die ursprünglich geplanten Meilensteine wurden schnell nach den Erkenntnissen aus Abschnitt 5.1.1 über Bord geworfen. Wir merkten schnell, dass wenn wir uns immer nur auf eine Form konzentrieren zu engstirnig denken. Am Ende haben wir jetzt zwei Prototypen. Einmal unsere geplante Implementierung in Android und eine Implementierung in Python (siehe Tabelle 1). Die Pythonimplementierung geht aus unseren Versuchen in Abschnitt 4.2 und 5.2 hervor und ist zurzeit nur auf dem Desktop lauffähig. Beide besitzen die Funktion Höhe und Breite eines Objektes mithilfe des Kamerabilds und einem Aruco zu bestimmen. Bei Beiden ist die Maßbestimmung noch sehr ungenau und verbesserungswürdig. Die Abweichungen von mehreren Millimetern macht die Maßbestimmung aktuell noch unbrauchbar. Es ist noch nicht möglich die Längen einzelner Kanten zu bestimmen, aber der Grundstein dafür wurde durch Ermitteln des Pixel-zu-Zentimeter-Werts gelegt. In beiden Implementierungen können die Konturen aus einer 3D-Datei ausgewertet werden und auf das Kamerabild gelegt werden. Wir können bereits sagen, dass nur die Konturen nicht ausreichend sind, um den Nutzer zu vermitteln, wie er die Kamera auf das Objekt zurichten ist. Es wäre besser alle zusehenden Kanten des Objekts auf das Kamerabild zulegen. Das Einzeichnen der Schablone ist in Android, wie in Abschnitt 5.1.2 bereits erklärt, nicht gut umgesetzt und war angesichts der näher rückenden Deadline eine Verzweiflungstat. Die Umsetzung mit Python aus einer *.stl*-Datei entstand erst kurz vor Ende und ist noch nicht ausgereift. Die Schablone wird in der Pythonimplementierung noch nicht mittig gelegt und das Soll-Bild im Hintergrund wird aktuell noch nicht optimal von den OpenCV Funktionen erkannt.
Im Projekt wurde noch nicht berücksichtigt wie ein automatisierter Soll-Ist-Abgleich stattfinden kann. Aktuell benötigen wir noch die Unterstützung des Menschen hinter dem Smartphone, dieser muss prüfen, ob das Objekt in die Schablone passt. Die Maße müssen manuell abgeglichen werden.
Unser gesetztes Ziel Möglichkeiten einer Qualitätskontrolle mit dem Smartphone zu erörtern

haben wir erfüllt. Aus unserer Sicht ist, dies aber nicht im uns gewünschten Umfang. Wir haben aus Zeitgründen nicht mehr geschafft uns damit zu beschäftigen wie nun ein automatisierter Abgleich zwischen soll und ist funktionieren könnte. Beide Implementierung sind nicht optimal und weit davon entfernt Marktfähig zu sein. Mit beiden Implementierung ist es dennoch möglich anschaulich unser entwickeltes Konzept (siehe Abschnitt 2) zu zeigen.

6.3 Ausblick

Mit sowohl der Python als auch der Implementierung für Android zeigen wir zwei Wege auf, die es sich lohnen könnte weiter zu gehen. Hauptaufgabe wäre es bei Beiden zunächst die Generierung des Ist-Bildes zu optimieren.
Wenn man es schaffen würde mit Kivy (siehe Abschnitt 3.2) zugriff auf das Kamerabild zu bekommen um es dann mit OpenCV auszuwerten, könnte man unser Pythonimplementierung als Smartphoneapp nutzen.
Für einen Soll-Ist-Vergleich mit der 3D-Datei könnte man unter anderem die Konturen auf Soll und Ist-Bild vergleichen. Bei einem Abgleich müsste man wahrscheinlich eine gewisse Abweichung berücksichtigen, da der User nie die Kamera zu 100% genau auf das Objekt richten kann. Für ein ermitteln der Soll-Maße müsste man noch etwas gesondert implementieren. Da weder *.stl* noch *.obj* eine Maßeinheit enthalten, müsste man erfassen wie die abstände zwischen den Koordinaten interpretiert werden sollen. Bereits erwähnte mögliche Feature für die App währen: eine gezielte Führung des Nutzers durch alle Perspektiven, die 3D-Datei wird von einem Server geladen, auf dem der technische Zeichner speichert seine Daten speichert und die App könnte automatisiert den Ausschuss festhalten.
Eine Übertragung auf andere Fertigungsvorgänge außerhalb vom Kunststoff 3D-Druck ist möglich. Je nach Werkstoff kann es zu neuen Problemen führen. So sind transparente und reflektierende Objekte schwerer auf einem Kamerabild auszuwerten.
Wir haben zwei Prototypen, die ein Konzept für zukünftige Nutzer verdeutlichen können. Mit diesen Prototypen könnten wir nun mit Nutzern in Kontakt treten, um weitere Ideen und Verbesserungen zu sammeln.

Anhang

Git Repositorys

`https://git.thm.de/mbbl33/quality-control-app`	Implementierung für Android
`https://git.thm.de/bozg22/i_4.0_quality_control_python`	Pythonimplementierung

Tabelle 1: Git Repositorys

Anmerkung zur Androidimplementierung: das Projekt ist wie in Abschnitt 4.3 umgezogen, daher sind ältere Commit-Nachrichten verloren gegangen. Die Inhalte der vorherigen der vorherigen Commits wurden händisch in das Projekt kopiert und mit dem initialen Commit im neuen Projekt gepusht.

Anmerkung zur Pythonimplementierung: diese wahr eigentlich nur zum Testen gedacht. Wir halten es dennoch für erwähnenswert.

Allgemeine Anmerkung: wie in der Ausarbeitung erwähnt hatten wir häufig Probleme lauffähige Sachen zu bekommen. Da wir keinen kaputten Code pushen wollten, hatten wir nur selten etwas 'commit-würdiges'. Zusätzlich möchten wir anmerken, dass wir viel Pair-Programming via Discord-Bildschirmübertragung und Code-with-me gemacht haben, daher tauchen in den beiden Repositorys nur einzelne Namen bei den Commits auf.

Bilder der 3D-Testdateien

Abbildung 22: Testdatei Bird

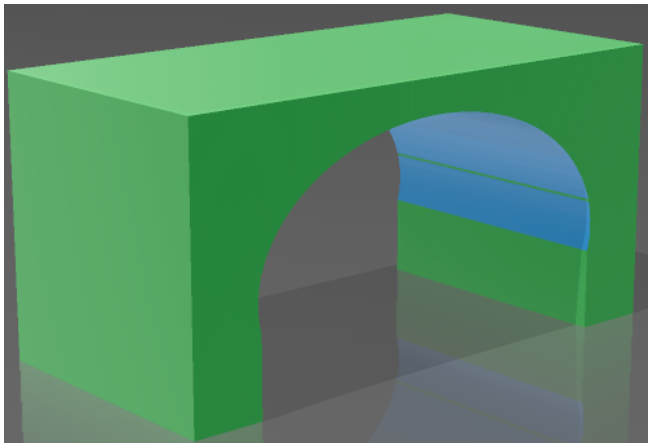

Abbildung 23: Testdatei Bruecke

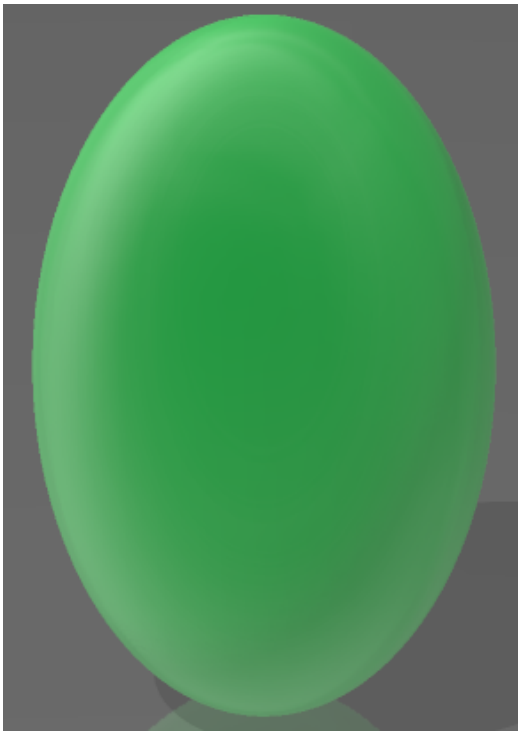

Abbildung 24: Testdatei Ei

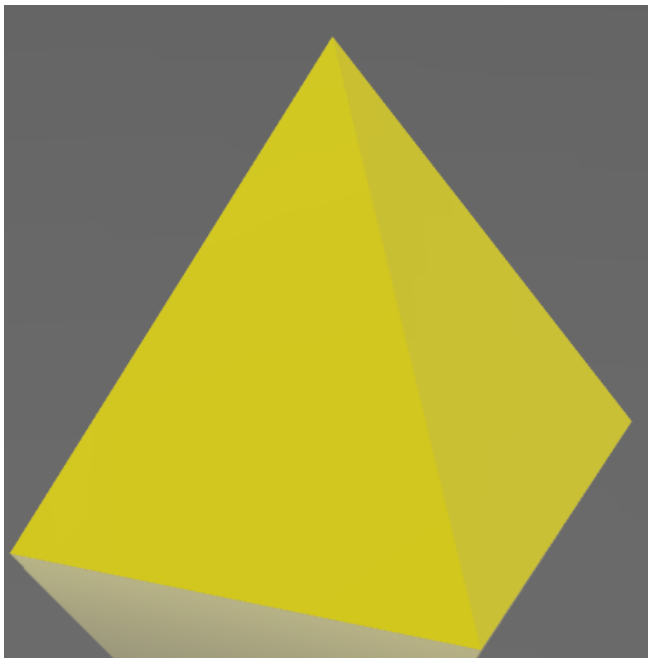

Abbildung 25: Testdatei Pyramide

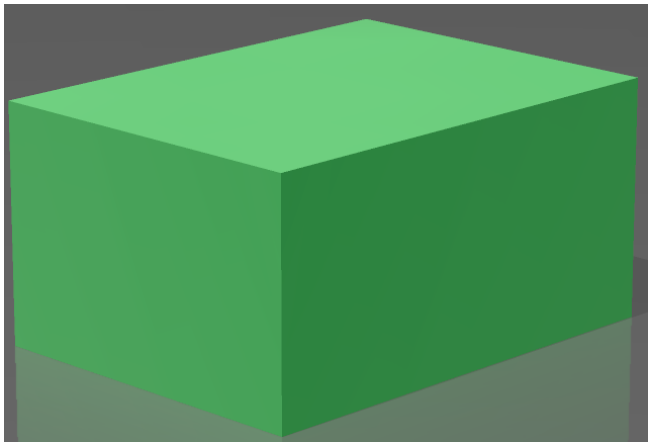

Abbildung 26: Testdatei Quader

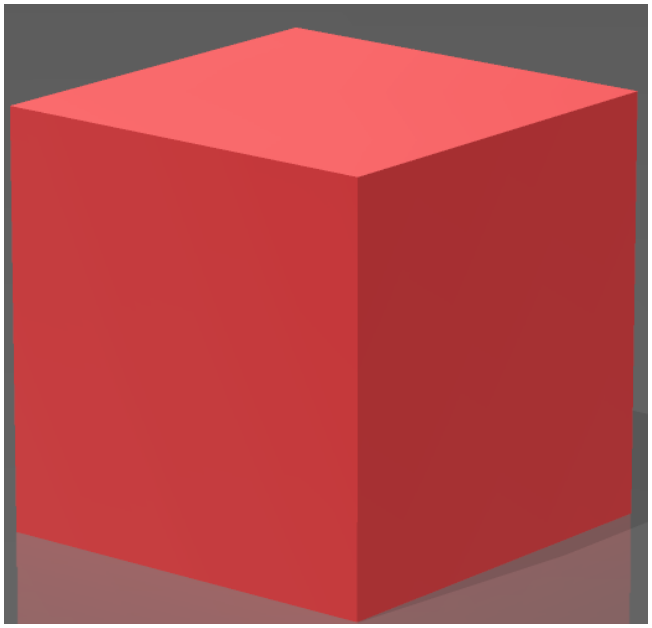

Abbildung 27: Testdatei Wuerfel_20mm

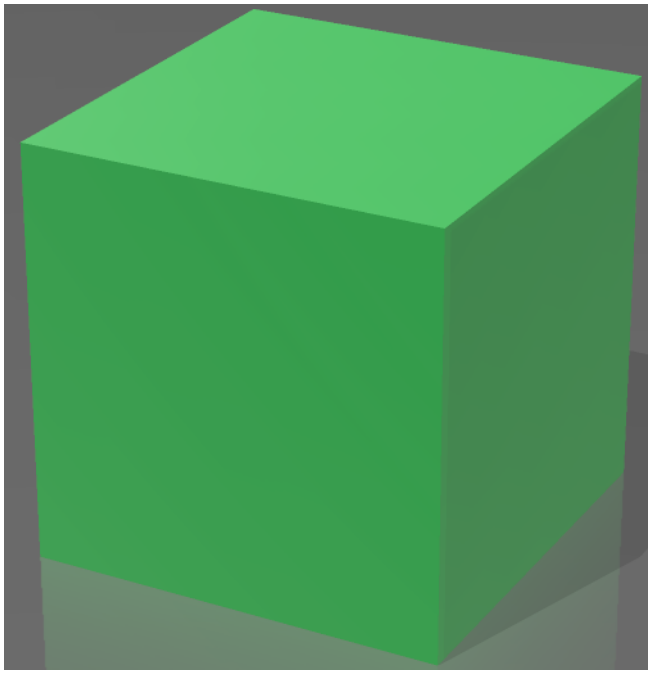

Abbildung 28: Testdatei Wuerfel_40mm

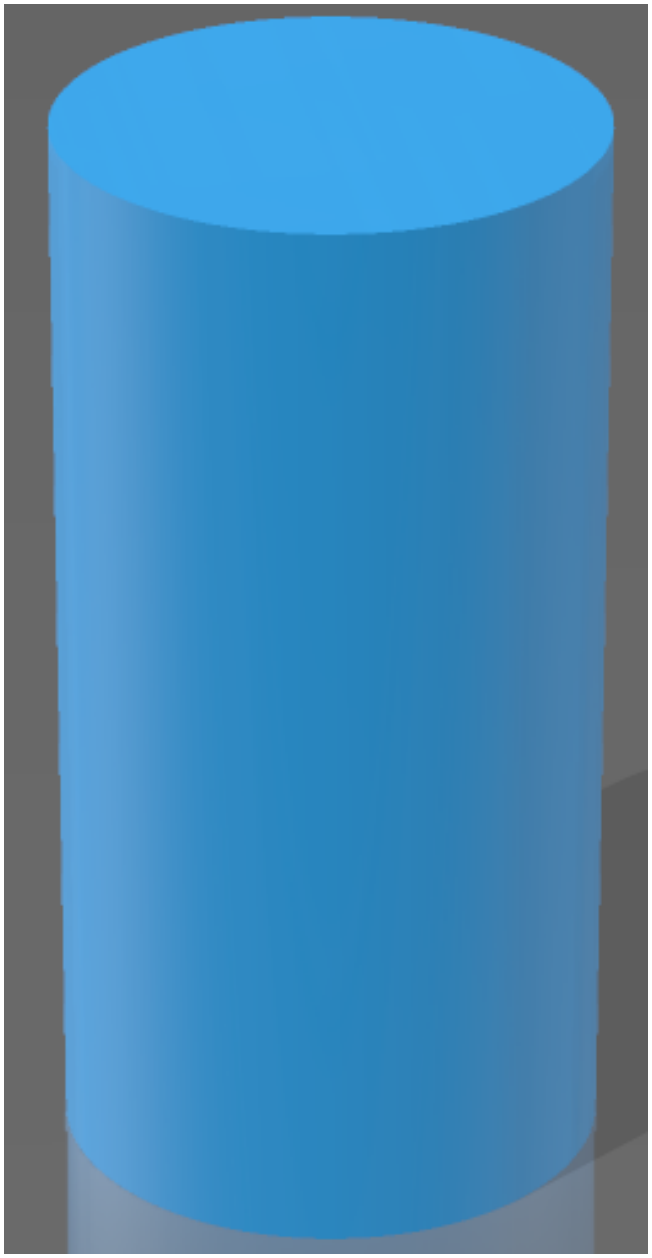

Abbildung 29: Testdatei Zylinder

Abbildungsverzeichnis

Literatur

[1] "Qualitätsmanagementsysteme - grundlagen und begriffe," Norm DIN EN ISO 9000, Nov. 2015.

[2] "Was ist xamarin?" https://learn.microsoft.com/de-de/xamarin/get-started/what-is-xamarin, 03-01-2023.

[3] "Was ist .net maui?" https://learn.microsoft.com/de-de/dotnet/maui/what-is-maui?view=net-maui-7.0, 03-01-2023.

[4] "Kivy," https://kivy.org/, 03-01-2023.

[5] "Opencv for android (java) youtube tutorial," https://youtu.be/imTTaZTSVQk, 03-01-2023.

[6] "Measure the size of an object — with opencv, aruco marker and python," https://youtu.be/lbgl2u6KrDU, 03-01-2023.

[7] "Measure the size of an object — with opencv, aruco marker and python," https://pysource.com/2021/05/28/measure-size-of-an-object-with-opencv-aruco-marker-and-python/, 03-01-2023.

[8] "Opencv android — easy way to integrate opencv into your android project via gradle." https://github.com/QuickBirdEng/opencv-android, 03-01-2023.

[9] "Processing," https://processing.org/, 10-01-2023.

[10] "Processing for android," https://android.processing.org/, 10-01-2023.

[11] "Processing for android - developing with android studio," https://android.processing.org/tutorials/android_studio/index.html, 11-01-2023.